Robust Policies for Transfer Learning in Robots

Rishhanth Maanav V [*]
Indian Institute of Technology Madras
Chennai, India
`ee16b036@smail.iitm.ac.in`

Siddharth Nayak[*]
Indian Institute of Technology Madras
Chennai, India
`ee16b073@smail.iitm.ac.in`

Abstract

Deep neural networks serve as function approximators with high complexity, and have enabled reinforcement learning agents to be used for a wide range of tasks, such as games and robotic control. An agent tries to learn policies and value functions through trial and error by interacting with an environment until it converges to an optimal policy. Robustness and stability are quite critical in reinforcement learning; however, neural networks are vulnerable to noise from unexpected sources and are not likely to withstand disturbances. Also, whenever robots are trained as reinforcement learning agents, they are generally either initialized with hand-made policies or policies from simulated agents. The simulated agents, however may not generalize well to real world due to scenarios which may not be captured while training. Transfer learning may not be efficient enough for fast convergence on robots due to these factors. In this work, we evaluate two robust reinforcement learning algorithms and compare them on different amounts of perturbations. The main idea drawn from this comparison is that adversarial examples can be used to choose perturbations while training an agent. Code for the experiments can be found at https://github.com/nsidn98/Robust-Reinforcement-Learning

1 Introduction

Deep neural networks, due to their high learning capacity, have been used extensively as function approximators in reinforcement learning agents. However, with deep networks, the need for extensive data arises. Obtaining data from real-world environments is challenging and difficult. Hence, simulations are used to model the real-world environment and train RL agents. However it is not possible to model and capture all the fine-grained details of the real world in the simulated environments. When such simulation-trained RL agents are transferred for fine-tuning in real-world, they might face scenarios which they have not been exposed to in simulations. As a result, convergence of the policy while fine-tuning would be slow. Essentially, data-intensive neural networks might not generalise well outside training distribution unless the training distribution is a good representative of the true distribution. In the case of simulations, clearly the training data fails to capture certain scenarios from the real-world making the networks not generalisable. Added to this is the varying amount of real-world noise to which neural networks, and hence RL agents using them, are vulnerable. Thus, robustness and stability are essential for any agent to function well. This calls for algorithms to capture this noise and adverse scenarios while training the RL agent. In this paper, we evaluate two robust reinforcement learning algorithms: 1) Robust Adversarial Reinforcement Learning (RARL) by Pinto et al.[1] and 2) Adversarial Robust Policy Learning (ARPL) by Mandlekar et al.[2]. We compare these two algorithms along with a naive algorithm which is described in section 3.3 and evaluate them with different amounts of perturbations while testing. We describe the RARL and ARPL algorithms in section 3. The experiments and results are described in section 4 and section 5 respectively.

[*]Equal Contribution

2 Background

2.1 Reinforcement Learning

Reinforcement learning tries to solve the sequential decision problems by learning from trial and error. Considering the standard RL setting where an agent interacts with an environment $\mathcal{E}$ over discrete time steps. In the time step t, the agent receives a state $s_t \in \mathcal{S}$ and selects an action $a_t \in \mathcal{A}$ according to its policy π, where $\mathcal{S}$ and $\mathcal{A}$ denote the sets of all possible states and actions respectively. After the action, the agent observes a scalar reward r_t and receives the next state s_{t+1}. The goal of the agent is to choose actions to maximize the cumulative discounted sum of rewards over time. In other words, the action selection implicitly considers the future rewards. The discounted return is defined as $R_t = \sum_{\tau=t}^{\infty} \gamma^{\tau-t} r_\tau$, where $\gamma \in [0, 1]$ is a discount factor that trades-off the importance of recent and future rewards.

RL algorithms can be divided into two main sub-classes: (1) Value-based and (2) Policy-based methods. In value-based methods, values are assigned to states by calculating an expected cumulative score of the current state. Thus, the states which get more rewards, get higher values. Policy-based algorithms like REINFORCE [3] and TRPO [4] try to learn a stochastic policy $\pi_\theta : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ that maximises R_t. Here θ denotes the parameters for the policy π. Generally the policy π is parameterized by a neural network.

2.2 Trust Region Policy Optimisation

TRPO is a policy gradient method which improves the stability and convergence of other general policy gradient and actor critic algorithms. TRPO modifies the unconstrained optimisation problem in policy gradient methods to a constrained optimisation problem as described below,

$$\begin{aligned} \underset{\theta}{maximize}\ \widehat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\widehat{A}_t\right] \\ \text{subject to } \widehat{\mathbb{E}}_t\Big[KL\big[\pi_{\theta_{old}}(.|s_t), \pi_\theta(.|s_t)\big]\Big] \le \delta \end{aligned} \tag{1}$$

Note that the expectation is taken over the trajectory sampled by the policy π_{old}

$\pi_\theta(a_t|s_t)$ policy parametrised by theta (network) ; a_t is the action taken at time 't' ; s_t is the state at time 't';
$\widehat{A}_t$is the estimated "generalized" advantage at time 't' where $\widehat{A}_t = Q(s_t, a_t) - V_\theta(s_t) + \tau\gamma\widehat{A}_{t+1}$;
with γ as the discount factor and τ as the generalization factor
Q is the return from a state obtained when a trajectory is sampled by π; $Q(s, a) = r + \gamma V_\theta(\overline{s}')$
In Actor-Critic models, the value function formulation of Q is used to generate an approximate return which is the sum of the current reward and the discounted expectation of the parametrised value of the next state.
If generalisation is not used, the advantage is the same as the TD error.
V_θ is the value function of a state which is parametrised by the critic network ;
δ is a hyperparameter ; KL stands for the Kullback-Leibler divergence

The maximization objective function has the ratio $\dfrac{\pi_\theta}{\pi_{\theta_{old}}}$ as the importance weight (from importance sampling). This is included because the maximization has to be done on sample trajectories generated by the policy parametrised by the new parameters θ whereas we sample only using the older policy $\pi_{\theta_{old}}$. Optimising this objective without any constraint is prone to numerical instability. Thus, TRPO makes sure that the policy is not moving too far from the current parameter θ_{old} by constraining the KL divergence to be bounded by δ as denoted above. For notational convenience,

$$L_{\pi_{\theta_{old}}}(\pi_\theta) = \widehat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\widehat{A}_t\right] \tag{2}$$

The constraint optimization problem is now the equivalent of solving,
$\underset{\theta}{maximize}\, L_{\pi_{\theta_{old}}}(\pi_\theta) - \beta.\overline{KL}_{\pi_{\theta_{old}}}(\pi_\theta)$;where β is a Lagrange Multiplier.
The matrix F which was described above is the Fisher Information matrix and g is the policy gradient.
The update step $(\theta - \theta_{old})$ is the natural policy gradient.

Algorithm 1 TRPO

```
1: described here is one step of the TRPO update rule :
2:         $\underset{\theta}{maximize}\, L_{\pi_{\theta_{old}}}(\pi_\theta) - \beta.\overline{KL}_{\pi_{\theta_{old}}}(\pi_\theta)$
3:         done by approximating $L$ linearly near $\theta_{old}$ and $KL$ quadratically near $\theta_{old}$
4:                 $\underset{\theta}{maximize}\, g.(\theta - \theta_{old}) - \frac{\beta}{2}(\theta - \theta_{old})^T.F.(\theta - \theta_{old})$
5:                 where $g = \frac{\partial}{\partial\theta} L_{\pi_{\theta_{old}}}(\pi_\theta)\Big|_{\theta=\theta_{old}}$ and $F = \frac{\partial^2}{\partial^2\theta}\overline{KL}_{\pi_{\theta_{old}}}(\pi_\theta)\Big|_{\theta=\theta_{old}}$
6:
7:                 Quadratic part of $L$ is negligible compared to KL term.
8:                 $F$ is positive semidefinite
9:                 $\theta \leftarrow \theta_{old} + \frac{1}{\beta}F^{-1}g$
```

The computation of the Fisher Information matrix (F) is expensive. It requires a second-order oracle to compute the hessian of the KL divergence term. Also, the parameter space is very high dimensional. Thus, optimizing the policy using the above described method is computationally expensive. So, we use conjugate gradient method to approximately obtain the update step without explicitly forming the Fisher Matrix F.

TRPO uses Generalized Advantage Estimation (GAE) for estimating advantages. These estimates require the future returns for computing the advantages. Thus, TRPO updates are done after the episodes are sampled completely in a fashion similar to Monte-Carlo methods.

3 Algorithms

We explain the Adversarial Robust Policy Learning (ARPL) and Robust Adversarial Reinforcement Learning (RARL) algorithms in this section.

3.1 Adversarial Robust Policy Learning

Adversarial Robust Policy Learning, is an algorithm to generate additive adversarial perturbations to states while training policy gradient algorithms on simulators. These perturbations generated by ARPL aim to capture the noise in the dynamics noise along with the process noise of the system. Dynamics noise is the error in measurement of physical quantities like mass, friction, etc. The dynamics noise influence the state, but only through a function which captures the dynamics of the system. Process noise is the noise which adds to the state directly.

ARPL uses a gradient based perturbation technique for adding adversarial perturbations to the states. There have been works in gradient based perturbations in the past. One of the most important contributions to adversarial perturbations is that of Fast Gradient Sign Method (FGSM) from Goodfellow et al. FGSM focuses on adversarial perturbations of an image input to a deep learning model where the change in each pixel is determined by the sign of the gradient over a defined loss function η and limited by a factor ε. The perturbations generated by FGSM could be extended to policy gradient algorithms where it is formulated as,

$$\delta = \varepsilon sign(\nabla_s \eta(\pi_\theta)) \qquad \text{(FGSM)} \tag{3}$$

where η is a loss function defined over the policy π parameterized by the policy network parameters θ and ε is a hyperparameter denoting the strength of the perturbations

This idea of FGSM is more suited for images while training deep learning models. ARPL extends this idea of gradient based perturbations of states to policy gradient algorithms.

In ARPL the perturbations of the states are computed as,

$$\delta = \varepsilon \nabla_s \eta(\pi_\theta) \qquad \text{(ARPL)} \tag{4}$$

where ε is a hyperparameter indicating the strength of the perturbation,η is a loss function over the policy π for perturbations. Note that this δ is not the same as that in TRPO. The policy π is parameterized by the policy network parameters θ

This loss function defined over the policy must be such that the added gradient perturbations to the states generally encourage the agent to take some corrective measures in its actions considering the noise present in the system. Note, that this loss function is different from the policy gradient loss and the value network loss. We'll describe the specific loss function which we use for our setting in section 4. At each timestep in an episode, we perturb the states using ARPL with a probability ϕ which is a hyperparameter.

Algorithm 2 Adversarially Robust Policy Learning (ARPL)

```
 1: Data: hyperparameters:$\phi, \epsilon, k$
 2: Initialize $\theta_0$
 3: for $i = 0, 1, 2, ..., max\_iter$ do
        // Perform k-Batch Rollout(s) with Adv. Perturb.
 4:     for each $t = 1, ..., Tk, \forall k$ do
 5:         Sample trajectory with policy $\pi(\theta_i)$: $\tau_{ik} = \{s_t, a_t\}_{t=0}^{T_{k-1}}$
 6:         Compute adv. perturb. $\delta = \epsilon\big(\nabla_{s_t}\eta(\pi_{\theta_i})\big)$
 7:         add perturbation($\delta$) with Bernoulli probability ($\phi$)
        // Batch Update to Policy Network
 8:     $\theta_{i+1} \leftarrow policy\_grad(\theta_i, \{\tau_{i1}, ...\tau_{ik}\})$ here the update step follows from TRPO
 9:     update params of value network using TD error
10: Result: Robust Policy $\pi$
```

3.2 Robust Adversarial Reinforcement Learning (RARL)

In Robust Adversarial Reinforcement Learning, we have two agents:

Protagonist: the one which tries to improve the overall performance for the task at hand.

Adversary: the one which tries to degrade the overall performance for the task at hand.

Both these agents play a zero-sum game. This adversary training helps the protagonist agent to explore more severe situations and thereby making it more adaptable to noisy environments.Both the protagonist and the adversary receive feedback from the environment and both give an action based on the observation. The difference is that the adversary takes an action that tries to fail the protagonist in it's task and make it get a lower reward. This is quite similar to the method used by Goodfellow et al.[5] in Generative Adversarial Networks (GAN) where two networks: Generator network and Discriminator Network compete with each other in a zero-sum game.

In RARL, there are 2 policy networks, one for the protagonist and the other for the adversary generating the policies π_p and π_a respectively. Similarly, there are two values networks for predicting the value function for the protagonist and the adversary V_p and V_a respectively. The action taken to sample the trajectory now is influenced by both the protagonist and the adversary. The exact mathematical formulation is described in the upcoming section.

3.2.1 The update rule

The Protagonist Policy π_p and the Adversary Policy π_a both participate in a zero-sum game. The actions of the protagonist and the adversary both affect the next state. The action taken by the algorithm to sample a trajectory is given as,

$$a = a_{protagonist} + D * a_{adversary} \tag{5}$$

where D is a scalar hyperparameter called the difficulty level. Here this parameter 'D' is set to confine the magnitude of the action from the adversarial agent. This is to ensure that the learning does not become unstable.

The expected reward $Q(s, a)$ is the cumulative discounted reward.

$$Q(s_k, a_k) = \mathbb{E}\Big(\sum_{t=k}^{T} \gamma^{t-k} * \big(r(s_t, a_t)\big)\Big) = \mathbb{E}\Big(r + \gamma * V(\bar{s})\Big) \tag{6}$$

The advantage function $A(s, a)$ is defined as the difference between the expected and the estimated reward: $A(s, a) = Q(s, a) - V(s)$. In the zero-sum game we have $r_{protagonist} = -r_{adversary} = r$. The update of the parameters are as follows:

$$\theta_p \leftarrow \theta_p + \alpha * A_{protagonist} * \nabla_{\theta_p} log(\pi_p) \tag{7}$$

$$\theta_a \leftarrow \theta_a + \alpha * A_{adversary} * \nabla_{\theta_a} log(\pi_a) \tag{8}$$

Here $A_{protagonist}$ is the difference between the discounted return ($Q(s, a)$) and the value network prediction ($V_{protagonist}(s)$). The discounted return is approximated by the sum of the current reward from the trajectory and the discounted expectation of the value of the next state predicted by the value network. The same applies to the adversary also but here the rewards are negated and hence the discounted return is also negated. For TRPO the advantages are generalized estimates and not the vanilla advantage where $A_t^{gen} = A_t + \gamma * \tau * A_{t+1}^{gen}$ with γ as the discount factor, τ as the generalization factor (GAE factor) and A_t is the vanilla Advantage at time t.

While testing, only the protagonist policy is used to sample actions.

Algorithm 3 Robust Adversarial Reinforcement Learning (RARL)

```
1: Data: hyperparameters:$D$
2: Initialize $\theta_p$ and $\theta_a$ for the policy networks
3: Initialize $w_p$ and $w_a$ for the value networks
4: for $i = 0, 1, 2, ..., max\_iter$ do
       perform k-batch rollouts
5:     for $t = 1, 2, 3...max\_timesteps$
           take action $a = a_{protagonist} + D * a_{adversary}$
6:     Update params $\theta_p$ and $\theta_a$ using TRPO update rule with rewards as R and -R respectively.
7:     Update params $w_p$ and $w_a$ with the Temporal Difference error at every step
```

3.3 Naïve and Vanilla Algorithms

We define a naïve and vanilla model to demonstrate the effectiveness of ARPL and RARL under adversarial conditions. In the vanilla model, we do not perturb the states while training. In essence the agent does not know anything about the perturbations. In the naïve model, we perturb the states of the agent randomly as:

$$s_t \leftarrow s_t + \alpha * x \tag{9}$$

where α is a hyperparameter which we set to 0.1, while training and $x \sim \mathcal{N}(0, 1)$ is a random number sampled from a Gaussian distribution with zero mean and unit standard deviation.
Note that the algorithms: ARPL, RARL and Naïve algorithms serve as additives to the vanilla TRPO algorithm. Thus, they can be extended to any other policy gradient algorithm.

4 Experiments

We demonstrate the effectiveness of ARPL and RARL under different strengths of perturbations through experiments in this section. For all our experiments we use TRPO as the base algorithm and we build the robust algorithms on top of it. TRPO is an actor-critic algorithm [6] and consists of two networks, one parameterising the policy and the other parameterising the values of state action pairs. As the action space is continuous, we let the agent predict the mean and standard deviation. The action is sampled from a gaussian with mean and standard deviation as predicted by the actor. The policy network is a 3-layered linear network with 64 nodes in both the hidden layers. The output layer depends on the action dimension of the environment and the input layer depends on the state dimension of the environment. All the layers have $tanh$ activations. The value network is same as the policy network except for the output layer which just has on node. The policy and the value network remain the same across all experiments with configurations as described earlier. The hyperparameters used for TRPO for all the agents are as follows:

- Discount Factor γ = 0.995
- Generalisation Factor for GAE in TRPO τ = 0.97
- L2-Regularisation rate λ = 0.001
- KL divergence constraint bound for TRPO δ = 0.01
- Damping coefficient for Conjugate Gradient method = 0.1

4.1 ARPL

In ARPL, we train the agent while perturbing the states as seem before in Eqn[4]. To trina ARPL, we use $\eta(\pi_\theta)$ as the squared $mathcal L_2$ norm of the mean vector predicted by the policy network. i.e if for an n-dimensional action $\mu = [\mu_1, \mu_2 \cdots \mu_n]$ is the mean vector predicted by the policy network, we use $\|\mu\|_2^2$ for the ARPL loss function. The intuition behind the loss is that, the state perturbed by the positive gradient of this loss will result in a higher value of the mean predicted by the policy network. Hence, by perturbing the state by this method, we trick the agent to believe that the perturbed state gave the same reward as the previous unperturbed state with the same action. Thus, the agent is encouraged to apply higher valued actions. We use perturbation strength $\epsilon = 0.1$ while training. Also, we used a curriculum learning approach to increase the probability ϕ of adding perturbations. We varied ϕ from 0.0 to 0.1 in 1000 iterations, increasing it every 100 iterations. This was done to maintain stability while training.

4.2 RARL

In RARL, both the protagonist and adversary have the same policy and value network configuration as mentioned before. The difficulty level D in Eqn[5] was set to 0.1 for stability in training. To improve stability, we only train the protagonist for the first 50 iterations. The protagonist and the adversary are then trained together at every iteration starting from the 50th iteration.

5 Results

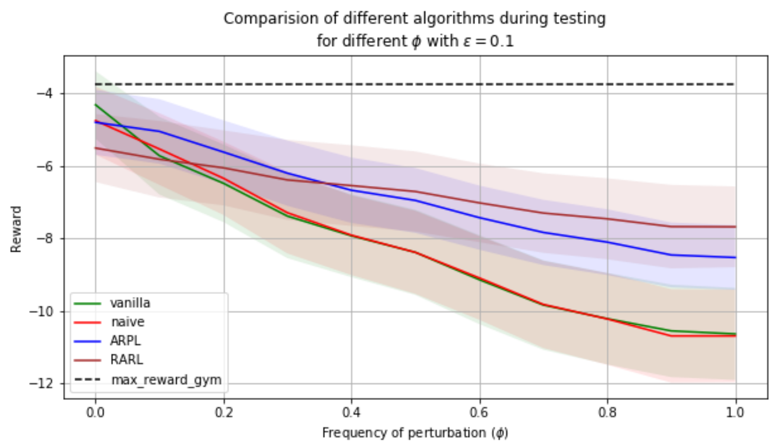
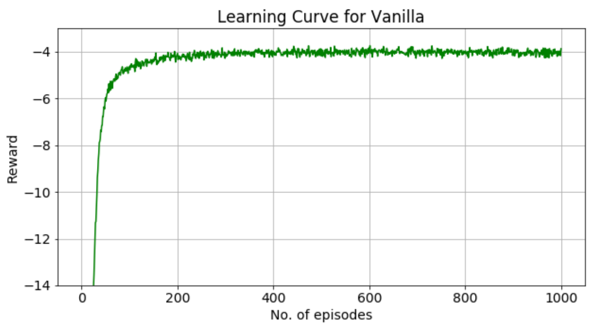
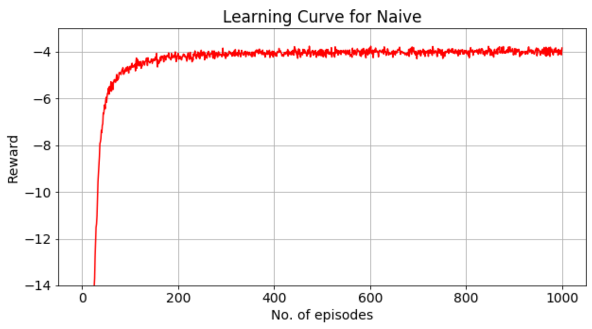


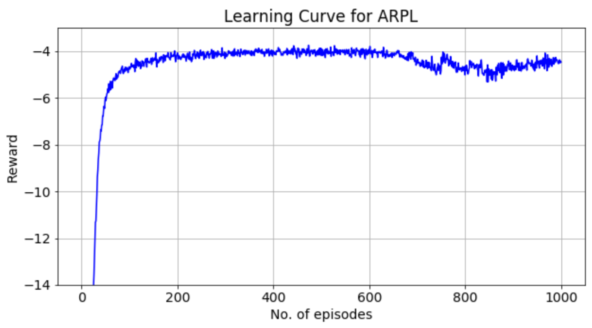
Figure 2: Performance of agents for the Reacher Environment with $\varepsilon = 0.1$ (test time) averaged over 1000 episodes



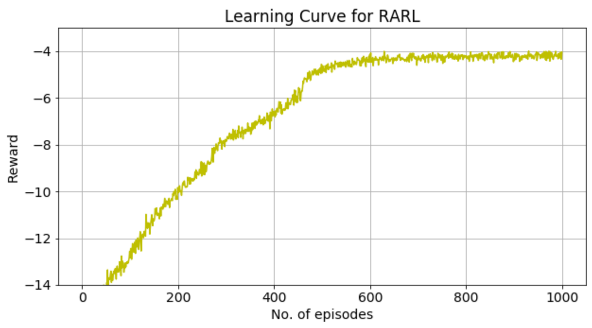
(a) Learning Curve for Vanilla



(b) Learning Curve for Naïve



(c) Learning Curve for ARPL



(d) Learning Curve for RARL

Figure 1: The Learning curves for the four different methods for the Reacher Environment. For ARPL, the rewards saturate after 200 episodes and then after 600 episodes it decreases slightly. This is because of the curriculum learning method used while training. i.e. the frequency of perturbations has increased. But the reward does increase again after 900 episodes. In RARL the saturation of the reward is slower than all other three algorithms because of the presence of the adversary agent even in the initial stages. The Naïve and the Vanilla network almost have the same learning curve.

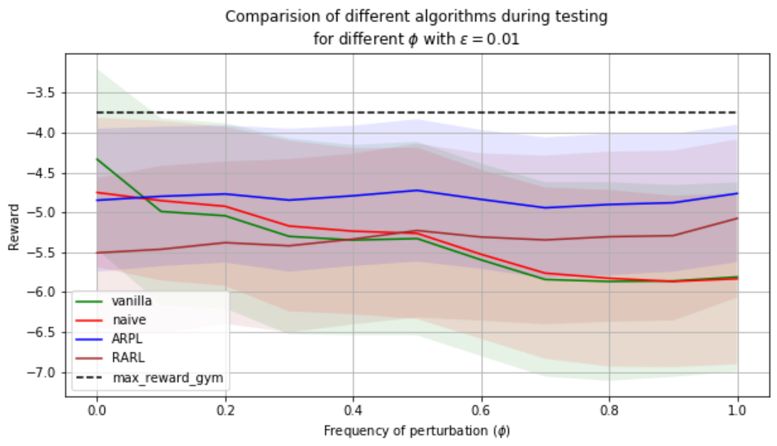


Figure 3: Performance of agents for the Reacher Environment with $\varepsilon = 0.01$ (test time) averaged over 1000 episodes

Based on the empirical results obtained from the evaluation of the different robust algorithms on the Reacher environment, we can draw some conclusions on their performance.

Average rewards for 1000 episodes for $\varepsilon = 0.1$ while testing				
ϕ	Vanilla	Naive	ARPL	RARL
0	-4.31 ± 1.85	-4.75 ± 1.87	-4.79 ± 1.79	-5.50 ± 1.87
0.1	-5.72 ± 2.13	-5.53 ± 1.97	-5.04 ± 1.77	-5.81 ± 2.10
0.2	-6.47 ± 2.13	-6.34 ± 2.01	-5.61 ± 1.74	-6.05 ± 2.06
0.3	-7.38 ± 2.31	-7.29 ± 2.24	-6.19 ± 1.79	-6.38 ± 2.20
0.4	-7.92 ± 2.27	-7.90 ± 2.19	-6.67 ± 1.80	-6.54 ± 2.22
0.5	-8.38 ± 2.34	-8.38 ± 2.28	-6.95 ± 1.78	-6.70 ± 2.20
0.6	-9.13 ± 2.44	-9.08 ± 2.30	-7.42 ± 1.76	-7.01 ± 2.16
0.7	-9.83 ± 2.46	-9.81 ± 2.40	-7.83 ± 1.79	-7.29 ± 2.18
0.8	-10.21 ± 2.52	-10.21 ± 2.52	-8.10 ± 1.81	-7.45 ± 2.22
0.9	-10.54 ± 2.54	-10.68 ± 2.57	-8.45 ± 1.78	-7.67 ± 2.29
1.0	-10.63 ± 2.53	-10.68 ± 2.55	-8.52 ± 1.78	-7.67 ± 2.22

Table 1: Comparison of average rewards obtained while testing for different algorithms with $\epsilon = 0.1$

Average rewards for 1000 episodes for $\varepsilon = 0.01$ while testing				
ϕ	Vanilla	Naive	ARPL	RARL
0	-4.33 ± 2.25	-4.75 ± 1.87	-4.84 ± 1.79	-5.50 ± 1.87
0.1	-4.98 ± 2.33	-4.85 ± 1.99	-4.79 ± 1.74	-5.46 ± 2.08
0.2	-5.04 ± 2.31	-4.92 ± 1.99	-4.76 ± 1.71	-5.37 ± 2.03
0.3	-5.30 ± 2.45	-5.17 ± 2.13	-4.84 ± 1.78	-5.41 ± 2.16
0.4	-5.34 ± 2.38	-5.23 ± 2.08	-4.79 ± 1.75	-5.33 ± 2.13
0.5	-5.32 ± 2.42	-5.26 ± 2.14	-4.72 ± 1.78	-5.22 ± 2.17
0.6	-5.59 ± 2.40	-5.52 ± 2.15	-4.83 ± 1.73	-5.30 ± 2.09
0.7	-5.84 ± 2.43	-5.76 ± 2.14	-4.94 ± 1.76	-5.34 ± 2.10
0.8	-5.86 ± 2.48	-5.82 ± 2.20	-4.90 ± 1.77	-5.30 ± 2.12
0.9	-5.85 ± 2.40	-5.86 ± 2.14	-4.87 ± 1.72	-5.29 ± 2.12
1.0	-5.80 ± 2.36	-5.83 ± 2.13	-4.76 ± 1.71	-5.07 ± 1.97

Table 2: Comparison of average rewards obtained while testing for different algorithms with $\epsilon = 0.01$

5.1 Perturbations with $\varepsilon = 0.1$

As seen in Fig 2, at lower frequencies of perturbations all four algorithms perform comparably well. As the frequency of perturbation is increased, the Vanilla and the Naive algorithm degrade in their performances. Whereas, ARPL and RARL degrade much lesser compared to Naive and Vanilla. This shows that the ARPL and RARL are quite robust to noise when compared to Vanilla and Naive. Another peculiar thing about the graph is that ARPL performs slightly better than RARL at smaller frequencies, but RARL is better at higher frequencies of perturbation. This is because while training RARL always has perturbations (adversary agent) while training whereas in ARPL we have curriculum learning, where the frequency of perturbations keeps on increasing with number of episodes while training. When there is no perturbation, RARL performs worse than others. This is because, RARL gets higher penalty (higher negative reward) for taking unnecessary actions to counter the non-existent perturbations. Also, ARPL does better than RARL in this case because the ARPL agent has perturbations included in the state while the RARL agent has the adversary perturbing its action while training. Thus, the RARL agent assumes the presence of the adversarial agent at all time instances and overcompensates for its effect even when it does not exist.

5.2 Perturbations with $\varepsilon = 0.01$

As seen in Fig 3, at lower frequencies we have almost comparable performances in all four algorithms. As the frequency is increased the Vanilla and the Naive algorithm degrade in their performance. In ARPL the performance is almost constant with increasing frequency in perturbation. RARL increases slightly in performance as the frequency of perturbation is increased. This serves as an evidence for the robustness of RARL and ARPL. However, RARL performs worse than the other three algorithms when there is no perturbation due to reasons seen in the previous subsection. In this case the strength of perturbation ε is only one-tenth of the previous case. Hence, the performance of all four algorithms is generally better in this case than the previous case.

As a final conclusion, both ARPL and RARL are robust to noise when evaluated with perturbations/noise in the states. RARL works well (compared to ARPL) when we know that frequency of noise occurence and the strength of noise in the system is high.But ARPL works better (when compared to RARL) when we know that the frequency of noise occurrence and its strength in the system is low.

6 Conclusion and Future Works

In this paper we presented the effectiveness of two robust reinforcement learning algorithms: RARL and ARPL with different amount of perturbations while testing. Future work includes working on a model where we combine both ARPL and RARL. In this model we perturb the state, as done in ARPL and perturb the action with an adversary agent, as done in RARL. This kind of dual perturbation may include the benefits of both ARPL and RARL.

References

[1] Lerrel Pinto, James Davidson, Rahul Sukthankar, and Abhinav Gupta. Robust adversarial reinforcement learning. *CoRR*, abs/1703.02702, 2017.

[2] Ajay Mandlekar, Yuke Zhu, Animesh Garg, Fei Fei Li, and Silvio Savarese. Adversarially robust policy learning: Active construction of physically-plausible perturbations. pages 3932–3939, 09 2017.

[3] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pages 1057–1063, Cambridge, MA, USA, 1999. MIT Press.

[4] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[5] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 2672–2680, Cambridge, MA, USA, 2014. MIT Press.

[6] Vijaymohan Konda. *Actor-critic Algorithms*. PhD thesis, Cambridge, MA, USA, 2002. AAI0804543.